JOURNAL OF COMPUTER AND SYSTEM SCIENCES 20, 203–218 (1980)

On Formulating Simultaneity for Studying Parallelism and Synchronization*

RAYMOND E. MILLER

Mathematical Sciences Department, IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598

AND

CHEE K. YAP†

Department of Computer Science, University of Southern California, Los Angeles, California 90007

Received November 13, 1979

1. INTRODUCTION

When studying parallel computation and synchronization one is faced with the problem of modeling the simultaneous execution of processes. Although there has been a multitude of formal means for representing such problems [2, 6, 9, 10, 14–16], invariably, when all the other complexities of the models have been stripped away, the parallelism or synchronization is studied via *sequences of events*. Thus in Petri nets [10] we have "firing sequences," in parallel program schemata [6] we have "$\mathscr{S}$-computations," in the system of processes model [9] we have "timings," and in the analysis of the mutual exclusion problem [14] we use "computation sequences." In all of these studies simultaneity of events is not studied directly. Rather, it is represented by the interleaving of the separate events into sequences, and by studying properties of the set of all such sequences.

This basic paradigm for simultaneity may be modeled thus: Let C be a set, usually called the *instructions*. Σ is a set of finite or infinite strings over C, called the *computation sequences*. Then two instructions, $a, b \in C$ are *simultaneous* if $\alpha ab\beta$ and $\alpha ba\beta$ are sequences in Σ, for some sequences α, β. It is very convenient to represent simultaneity in terms of sequences within these formulations. Sequences are familiar objects of study and can be comfortably analyzed and manipulated. Also, this means of representing simultaneity is often sufficient for studying the properties of interest. Nevertheless, it should be clear that interleaving alone is a weaker notion than one that allows the possibility of simultaneity. We shall see this more clearly later.

* An early version of this paper was presented at the Tenth Annual ACM Symposium on Theory of Computing, May 1, 1978, and appears in that Proceedings.

† This work was done while this author was a summer employee at the IBM Thomas J. Watson Research Center.

0022-0000/80/020203–16$02.00/0

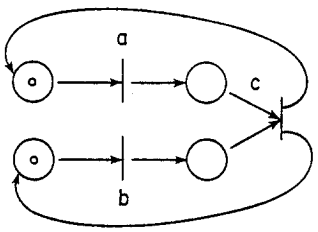

FIGURE 1

As an example of simultaneity, in Fig. 1, we have a Petri net whose firing sequences (represented as sequences over the transitions $\{a, b, c\}$) are the set Σ, which are initial segments of words of the form $\alpha_1 c \alpha_2 c \alpha_3 c \ldots$, where $\alpha_i \in \{ab, ba\}$. Clearly a and b are simultaneous.

One could make the case that all the different approaches to simultaneity in the literature reduce to being alternative ways for finitely describing the (potentially) infinite set Σ. This bold claim is an oversimplification, although as a caricature it helps us to see the basic connections between the alternative models.

We have no objection to the basic paradigm outlined above. But we take exception to the usual interpretation of the set C as "instructions," where the instructions are assumed to be "unanalyzable wholes." The typical justification for this attitude runs as follows. If simultaneous instructions from different processes are allowed, as, for example, where one instruction is assigning a new value to a common variable while the other is reading from that variable, then the result would be indeterminate. Thus, it is meaningless to discuss "true" simultaneity in general. Instead it is postulated that an instruction has to be executed to completion before another instruction from any other process can initiate. This is often called *indivisibility of instructions*.

Indivisibility of instructions, besides the fact that it prevents the embarrassing situation of simultaneous instruction execution, may be justified under the following two conditions:

(i) The instructions are sufficiently elementary that they take only one machine-cycle. We must assume here that interrupts are "inter-cycle," not "intra-cycle."

(ii) There is only one independent access channel to common data.

If either of these conditions breaks down, then indivisibility of instructions becomes an unrealistic assumption. The literature, however, frequently deals with nonelementary instructions. The P–V primitives of Djikstra [3, 4] and their generalizations are prime examples of this. Even at the machine language level, some instructions are interruptible. For example, consider the Move Long instruction, MVCL R_1, R_2, in the IBM 370 (pp. 133–134, IBM System/370, Principles of Operation). Also, in recent years, there is an increasing trend toward modeling distributed computer systems in synchronization problems. With more than one access channel, even if instructions take one machine-cycle, we have no guarantee that instructions may not be simultaneous and cause indeterminacies. In the light of these arguments, it seems desirable to abandon a blind

assumption of instruction indivisibility and look at the problem in more detail. As a rare exception to the quick dismissal of simultaneity, these issues are discussed in some detail in the paper of Gilbert and Chandler [5]. In that sense [5] is closely related to this paper, yet the approach and the results are quite different.

2. A Form of Simultaneity

We shall illustrate how the Indivisibility of Instructions Assumption may be relaxed in a controlled and manageable way. Consider the system of two concurrent processes in Fig. 2.

At the termination of the two processes, it is easy to see that S has the value 0 or 1 depending on whether the execution sequence was (c_0^1, c_0^2) or (c_0^2, c_0^1) (we omit the *end* instructions), respectively. On closer examination, it appears that $S = 2$ may also be a reasonable outcome. This is shown using a device introduced by Pratt and Rivest [14]:

INITIALLY $S = 1$

(Process 1) c_0^1 : $S \leftarrow S + 1$
c_1^1 : *end*

(Process 2) c_0^2 : $S \leftarrow C$
c_1^2 : *end*

FIGURE 2

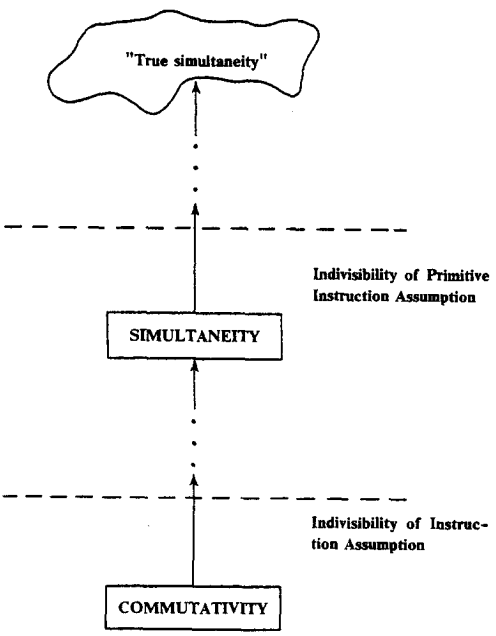

FIGURE 3

We say that c_0^1 should really be analyzed as two primitive instructions, $c_0^1 \equiv c_{01}^1; c_{02}^1$, where c_{01}^1: $U \leftarrow S$ and c_{02}^1: $S \leftarrow U + 1$. Here U is an "invisible" variable which is accessed by Process 1 only. Now if the execution sequence were $(c_{01}^1, c_0^2, c_{02}^1)$ then indeed $S = 2$.

What we do in this paper is to generalize the approach exemplified above. It consists of breaking down instructions into sufficiently simple "atomic acts." In this paper the atomic acts considered are at the level of reading or writing a single memory location. We call the actions at this level *Primitive Actions.* This seems to be a low enough level for our purpose, but even lower levels might be considered. We are not assuming bitwise actions in which, for example, in the previous example the final value of S could have been a bitwise mixture of the c_0^1 and c_0^2 actions. We call our assumption the *Indivisibility of Primitive Instructions Assumption* and the resulting formal model, the *Simultaneity Model.* In terms of the discussion in the introduction, we are interpreting C as the set of primitive instructions. In contrast, when the Indivisibility of Instructions Assumption is taken, the corresponding model is called the *Commutativity Model.* Figure 3 shows the conceptual relation between "True Simultaneity" (an intuitive and nebulous concept), the commutativity model, and our simultaneity model. The arrows indicate the path towards more accurate approximations of "True Simultaneity."

A natural question then arises: How much more general is the simultaneity model over the commutativity model? Our main theorem answers this question by giving restrictions under which simultaneity and commutativity are equivalent.

These restrictions deal with how simple the instructions in the processes must be. Therefore, if the instructions of the processes all satisfy these constraints, it is sufficient to study simultaneity via the more convenient commutative sequence methods. To conclude the paper we illustrate that when the constraints of the theorem are violated, then simultaneity leads to a wider class of behaviors than commutativity. Thus, the hypotheses of the theorem cannot be deleted, and in that sense the theorem is tight.

We feel that these results help clarify the apparent paradox caused by sidestepping the issue of simultaneity in models for parallelism and synchronization. It illustrates, within the confines of the particular formalism, when the commutativity approach is feasible. It also raises the question of how one might construct an appropriate theoretical framework to provide a useful hierarchy of synchronization protection, so that in practice one would only need to be concerned about correct synchronization at a high level rather than at the primitive instruction level indicated by our results.

In [11] we find a more extensive formulation for synchronization problems. Section 6 of [11] contains an early version of the results of this paper.

Notational Conventions

If A and B are sets then α: $A \rightarrow B$ represents a function α from A to B, that is, with domain A and range B. $\mathbf{X}_m A$ is the m-fold Cartesian product of A. $\mathbf{X}_{i=1}^m A_i$ is the Cartesian product of $A_1, \ldots, A_m$. We use $\bar{x} = \langle x_1, \ldots, x_n \rangle$ to denote n-tuples, and $\prod_i(\bar{x})$ to denote the projection function on the ith coordinate of the n-tuple; thus, $\prod_i(\langle x_1, \ldots, x_n \rangle) = x_i$. If $\bar{x} = \langle x_1, \ldots, x_n \rangle$ and $\bar{y} = \langle y_1, \ldots, y_m \rangle$ are tuples, then $\bar{x}; \bar{y}$

denotes the $(n + m)$-tuple $\langle x_1, \ldots, x_n, y_1, \ldots, y_m \rangle$ and x_0; $\bar{x}$ denotes the $(n + 1)$-tuple $\langle x_0, x_1, \ldots, x_n \rangle$. The set $\{1, 2, \ldots, n\}$ is denoted by $[n]$. $| A |$ is the cardinality of the set A and N is the set of natural numbers $\{1, 2, \ldots\}$. A sequence is denoted by $\bar{\sigma} = (\sigma_1, \ldots, \sigma_n) = (\sigma_i)_{i=1}^n$ or $\bar{\sigma} = (\sigma_1, \sigma_2, \ldots)$. If the members of a sequence belong to a set A, then we say that it is a sequence in A. Note that we have used "overbars," $\bar{\sigma}$ or $\bar{x}$, for sequences as well as tuples. This should not lead to any confusion. If $\bar{\sigma} = (\sigma_1, \ldots, \sigma_n)$ and $\bar{\tau} = (\tau_1, \ldots, \tau_m)$ then $\bar{\sigma}; \bar{\tau}$ is the sequence $(\sigma_1, \ldots, \sigma_n, \tau_1, \ldots, \tau_m)$. The empty set is denoted by $\varnothing$.

3. System of Processes

Our model of a "system of processes" has affinities with the model introduced by Lipton [9] and also that of Gilbert and Chandler [5]. Each process is deterministic and sequentially executes instructions. An instruction of a process determines two things: it computes new data values and specifies the next instruction of the process to be executed. There is a common data set accessed by processes in the system, and it is through this common data set that interactions between processes occur.

Definition 1. $\mathcal{D} = \langle \bar{d}_0, D \rangle$ is called the *data set* where $D = \mathsf{X}_{i=1}^m D_i$, each D_i is a set, $i = 1, 2, \ldots, m$, and $\bar{d}_0$ is an arbitrary element of D called the *initial data.*

Definition 2. A *process on* $\mathcal{D}$ is a 3-tuple, $P = \langle C, \lambda, \mu \rangle$, where:

(i) C is a finite set called the (instruction) *counter values* with two distinguished elements c_0 and c_h. c_0 is the *initial counter value* and c_h the *halt counter value.*

(ii) $\lambda: (C - \{c_h\}) \times D \to C$ is the *next instruction function.*

(iii) $\mu: (C - \{c_h\}) \times D \to D$ is the *data transformation function.*

Definition 3. A *system of n processes on* $\mathcal{D}$ is a set $\Sigma = \{P^i\}_{i=1}^n$, where each $P^i = \langle C^i, \lambda^i, \mu^i \rangle$, $i = 1, 2, \ldots, n$, is a process on $\mathcal{D}$, and $C^i \cap C^j = \varnothing$ for $i \neq j$.

We call P^i the ith *process.* We will use superscripts to denote the process being referred to. For example, $c_0{}^i$ and $c_h{}^i$ are the initial and halt counter values of the ith process. A typical member of $\mathcal{D}$ is $\bar{d} = \langle d_1, \ldots, d_m \rangle$. We take $\mathcal{D}$ and Σ to be fixed for the rest of this paper. Since $C^i \cap C^j = \varnothing$ for $i \neq j$, we may sometimes write $\lambda(c; \bar{d})$ or $\mu(c: \bar{d})$ instead of $\lambda^i(c; \bar{d})$ or $\mu^i(c; \bar{d})$ since i can be understood from c.

Each instruction of P^i is represented by a counter value in C^i. We will often call C^i the *instructions* of P^i. Each P^i begins its computation at the initial instruction $c_0{}^i$. The only communication between processes occurs through $\mathcal{D}$. No process may modify or read another process's counter value. Although $\mathcal{D}$ could include all the variables of interest in the computational, control, and interaction aspect of the processes, it is often convenient to consider $\mathcal{D}$ to be only the data used for process interaction.

In [11], we included the notion of system failure in the model. Although the results go through directly including system failure, for simplicity we omit this aspect of the model

here. In addition, the model here differs from [11, 12] by including a special *halt* instruction to allow more natural modeling of programs that terminate.

DEFINITION 4. An *instantaneous description* (i.d.) of Σ is an $(n + m)$-tuple, $I = \langle c^1,..., c^n, d_1,..., d_m\rangle$, where $c^i \in C^i$, $i = 1, 2,..., n$, and $\langle d_1,..., d_m\rangle \in D$. The *initial i.d.* of Σ is $I_0 = \bar{c}_0; \bar{d}_0$ where $\bar{c}_0 = \langle c_0^1, c_0^2,..., c_0^n\rangle$, and $\bar{d}_0$ is the initial data.

DEFINITION 5. Let $I = \bar{c}; \bar{d}, I' = \bar{c}'; \bar{d}'$ be i.d.'s of Σ, $i \in [n]$. The binary relation "$\rightarrow_{i,\Sigma}$" is said to hold between I and I', written $I \rightarrow_{i,\Sigma} I'$, iff

(i) $\prod_j(\bar{c}') = \prod_j(\bar{c})$, for $j = 1,..., n, j \neq i$,

(ii) $\prod_i(\bar{c}') = \lambda^i(\prod_i(\bar{c}); \bar{d})$,

(iii) $\bar{d}' = \mu^i(\prod_i(\bar{c}); \bar{d})$.

We then say that the instruction $\prod_i(\bar{c})$ *acted* in the transition $I \rightarrow_{i,\Sigma} I'$. We also say that the transition $I \rightarrow_{i,\Sigma} I'$ is *caused* by the ith process. Note that if $\prod_i(\bar{c}) = c_h^i$, i.e., the ith process has halted, then $\lambda^i(\prod_i(\bar{c}); \bar{d})$ is undefined so the ith process cannot cause any transition from I. We write $I \rightarrow_\Sigma I'$ iff $\exists i \in [n]$ such that $I \rightarrow_{i,\Sigma} I'$. The reference to Σ is usually omitted in $\rightarrow_\Sigma$ and $\rightarrow_{i,\Sigma}$. As usual, $\rightarrow^*$ and $\rightarrow_i^*$ is the reflexive transitive closure of $\rightarrow$ and $\rightarrow_i$. If $I_0 \rightarrow^* I_1$, where I_0 is the initial i.d., we say that I_1 is *reachable.*

DEFINITION 6. A sequence of i.d.'s $\mathscr{I} = (I_1, I_2,...)$ is called a *transition sequence* iff $\forall i \geqslant 1, I_i \rightarrow I_{i+1}$.

4. REALIZATIONS

If $\bar{\sigma} = (\sigma_1,..., \sigma_m)$ and $\bar{\tau} = (\tau_1,..., \tau_n)$ are sequences, then the *shuffle product* of $\bar{\sigma}$ and $\bar{\tau}$ is the set of all sequences of the type $\bar{\nu} = (\nu_1,..., \nu_{n+m})$ such that $\bar{\sigma}$ and $\bar{\tau}$ appear as disjoint subsequences of $\bar{\nu}$. For example, $\bar{\sigma} = (a, b, c), \bar{\tau} = (1, 2)$, then $(a, b, c, 1, 2), (a, b, 1, c, 2), (1, a, 2, b, c), (1, 2, a, b, c)$, etc. are elements of the shuffle product of $\bar{\sigma}$ and $\bar{\tau}$. We will not use shuffle products of sequences $\bar{\sigma}$ and $\bar{\tau}$ which contain common occurrences of letters. It is easy to extend the definition to give the shuffle product of a finite set of sequences.

The first step in our goal to formalize a notion of simultaneity is to define a class of atomic actions which could act in combination to simulate any instruction. If c is an instruction, we want to define a set of *primitive instructions belonging to c*, $A(c)$. If $\bar{\sigma} = (\bar{\sigma}_1,..., \bar{\sigma}_k)$ is some suitable ordering of the primitive instructions of $A(c)$ such that when each σ_i is executed in the indicated order, the result is the same as having executed c, we call $\bar{\sigma}$ a *simple realization* of c. If $\bar{\tau}$ is a simple realization of another instruction c', then the simultaneous execution of c and c' may be reflected in the shuffle product of $\bar{\sigma}$ and $\bar{\tau}$. We illustrate this basic strategy by considering an informal example:

$$c_0\text{: } do\ x \leftarrow f(t, x)\ then\ goto\ c_1\ od,$$

where

$$f(t, x) = 0 \quad \text{if} \quad t = 0$$
$$= x \quad \text{otherwise.} \qquad (*)$$

The variables x and t are assumed to range over the integers. The primitive instructions belonging to c_0 are of three sorts. First, there are the reading actions:

$$\mathbf{Rd}_{x,c_0}\colon x' \leftarrow x,$$

$$\mathbf{Rd}_{t,c_0}\colon t' \leftarrow t.$$

Here we regard x' and t' as *internal variables* seen only by the process of c_0 (hence x, t are also called *external variables*). In general for each external variable z we have an internal copy z'.

Second, we have the writing actions:

$$\mathbf{Wt}_{x,c_0}\colon x \leftarrow f(x', t').$$

The important thing to note is that $\mathbf{Wt}_{x,c_0}$ computes using internal variables.

Finally we have the atomic action corresponding to updating the program counter:

$$\boldsymbol{\lambda}_{c_0} : \textit{goto } c_1 .$$

Hence the set of primitive instructions belonging to c_0 is $A_1(c_0) = \{\mathbf{Rd}_{x,c_0}, \mathbf{Rd}_{t,c_0}, \mathbf{Wt}_{x,c_0}, \boldsymbol{\lambda}_{c_0}\}$. It is also easy to see the sequences $(\mathbf{Rd}_{x,c_0}, \mathbf{Rd}_{t,c_0}, \mathbf{Wt}_{x,c_0}, \boldsymbol{\lambda}_{c_0})$ and $(\mathbf{Rd}_{t,c_0}, \mathbf{Rd}_{x,c_0}, \mathbf{Wt}_{x,c_0}, \boldsymbol{\lambda}_{c_0})$ are two simple realizations of c_0. Of course, still other simple realizations are possible.

The alert reader may already note that we made two questionable tacit assumptions whose importance becomes apparent when one discusses concurrent processes. The first assumption is that the action c_0 need not "update" the variable t as it did for x. The primitive instruction in question is:

$$\mathbf{Wt}_{t,c_0}\colon t \leftarrow t'.$$

If this assumption is revoked, then the set of primitive actions belonging to c_0 ought to be $A_2(c_0) = A_1(c_0) \cup \{\mathbf{Wt}_{t,c_0}\}$. To see that this alternative definition of $A_2(c_0)$ leads to semantically inequivalent instructions, consider $\bar{\sigma} = (\mathbf{Rd}_{x,c_0}, \mathbf{Rd}_{t,c_0}, \mathbf{Wt}_{x,c_0}, \mathbf{Wt}_{t,c_0}, \boldsymbol{\lambda}_{c_0})$ as a simple realization of c_0. Let c_1 be the instruction

$$c_1\colon t \leftarrow t + 1.$$

It is not hard to see that if $\bar{\tau}$ is any simple realization of c_1, we can find $\bar{\nu}$ a member of the shuffle product of $\bar{\tau}$ and $\bar{\sigma}$ such that the final effect of executing $\bar{\nu}$ leaves t to have its original value (simply let $\mathbf{Wt}_{t,c_0}$ be the last writing action in $\bar{\nu}$. However, such a result

cannot occur if we take the primitive actions belonging to c_0 to be $A_1(c_0)$. This illustration should warn us about potential difficulties in formalizing a suitable notion of simple realizations.

Another tacit assumption in defining $A_1(c_0)$ is related to the semantics of $f(x, t)$: In sequential programming the definition given by (*) is adequate, but for our purposes, it is unclear whether the variable x should be updated or it should be "left alone" in case $t \neq 0$. In $\mathbf{Wt}_{x,c_0}$ we assumed x is always updated, but we could as well define:

$$\mathbf{Wt}_{x,c_0,t=0}: \quad \textit{if} \quad t' = 0 \quad \textit{then} \quad x \leftarrow 0 \; \textit{fi}.$$

The primitive instruction belonging to c_0 under this assumption is $A_3(c_0) = \{\mathbf{Rd}_{x,c_0}, \mathbf{Rd}_{t,c_0}, \mathbf{Wt}_{x,c_0,t=0}, \boldsymbol{\lambda}_{c_0}\}$.

Remark. The above do not exhaust the possibilities. One can imagine even more perverse semantics such as:

$$\mathbf{Wt}_{x,c_0,t=0\wedge x\neq 0}: \quad \textit{if} \quad t' = 0 \wedge x' \neq 0 \quad \textit{then} \quad x \leftarrow 0 \; \textit{fi}.$$

Both of the above ambiguities arise because in sequential programming, the function specification, i.e., input-output description, of an instruction is usually sufficient. But in concurrent programming, functional specification is unable to distinguish between a variable that is to be left "untouched" and one that is to be "recopied." Our solution is the introduction of ψ, a collection of predicates on D such that for each variable $i \in [m]$, $\psi_i \in \psi$. The predicate ψ_i is used to control the updating of variable i. Informally,[1]

$$\mathbf{Wt}_{i,c,\psi}: \quad \textit{if} \quad \psi_i(\bar{d}') \quad \textit{then} \quad d_i \leftarrow \prod_i (\mu(c, \bar{d}')) \; \textit{fi},$$

where $\bar{d}'$ is the internal copy of $\bar{d}$, $d_i = \prod_i (\bar{d})$. The set of primitive instructions belonging to c is now uniquely specified *up to a choice of* ψ, denoted by $A_\psi(c)$. For instance, in the case of $A_1(c_0)$ of the preceding example, ψ_x is **true** and ψ_t is **false**; in $A_2(c_0)$, both ψ_x and ψ_t are **false**; in $A_3(c_0)$, ψ_x is "$t = 0$" and ψ_t is **false**.

We now formalize the preceding discussion. From now on, $\tilde{C} = C^1 \times C^2 \times \cdots \times C^n$ and $\tilde{D} = \mathsf{X}_{n+1} D$. An *extended i.d.* is $\tilde{I} = \bar{c}; \langle \bar{d}_1, \ldots, \bar{d}_{n+1} \rangle \in \tilde{C} \times \tilde{D}$. We call $\bar{d}_i$ for $i \in [n]$ the *internal data* of the ith process, and $\bar{d}_{n+1}$ the *external data* (common to all processes). We regard a typical element of $\tilde{D}$ to be $\tilde{d} = \langle \bar{d}_1, \ldots, \bar{d}_{n+1} \rangle$, an $(n+1)$-vector whose components are m-vectors, rather than as an $(n+1) \cdot m$-vector. For any vector $\bar{x}$, we let $\bar{x}[y/i]$ denote the vector identical to x except that the ith component is replaced by y. We introduce the *extractor function* E: $\tilde{C} \times \tilde{D} \to \tilde{C} \times D$ given by $E(\bar{c}; \langle \bar{d}_1, \ldots, \bar{d}_{n+1} \rangle) = \bar{c}; \bar{d}_{n+1}$. Hence E simply ignores the internal data and extracts an i.d. from an "extended i.d.."

[1] The general notation of $\mathbf{Wt}_{i,c,\psi}$ we now adopt is not consistent with those of $\mathbf{Wt}_{x,c_0}$, $\mathbf{Wt}_{x,c_0,t=0}$, etc., of the introductory example. Some resemblance is clear.

DEFINITION 7. Let $i \in [m]$ and c be an instruction of process j. The following are *primitive instructions belonging to* c:

(i) $\boldsymbol{\lambda}_c$: $\tilde{C} \times \tilde{D} \to \tilde{C} \times \tilde{D}$ such that $\boldsymbol{\lambda}_c(\bar{c}; \langle \bar{d}_1, \ldots, \bar{d}_{n+1} \rangle)$ is undefined if $\prod_j(\bar{c}) \neq c$, otherwise $\boldsymbol{\lambda}_c(\bar{c}; \langle \bar{d}_1, \ldots, \bar{d}_{n+1} \rangle) = \bar{c}'; \langle \bar{d}'_1, \ldots, \bar{d}'_{n+1} \rangle$, where $\bar{d}'_l = \bar{d}_l$ for $l \in [n+1]$, and $\bar{c}' = \bar{c}[z/j]$, where $z = \lambda^j(c; \bar{d}_j)$.

(ii) $\mathbf{Rd}_{i,c}$: $\tilde{C} \times \tilde{D} \to \tilde{C} \times \tilde{D}$ such that $\mathbf{Rd}_{i,c}(\bar{c}; \langle \bar{d}_1, \ldots, \bar{d}_{n+1} \rangle)$ is undefined if $\prod_j(\bar{c}) \neq c$, otherwise $\mathbf{Rd}_{i,c}(\bar{c}; \langle \bar{d}_1, \ldots, \bar{d}_{n+1} \rangle) = \bar{c}'; \langle \bar{d}'_1, \ldots, \bar{d}'_{n+1} \rangle$, where $\bar{c}' = \bar{c}$, $\bar{d}'_l = \bar{d}_l$ for $l \neq j$, and $\bar{d}'_j = \bar{d}_j[z/i]$, where $z = \prod_i(\bar{d}_{n+1})$.

(iii) Let $\psi = \{\psi_i : i \in [m]\}$ be a set of predicates on D. $\mathbf{Wt}_{i,c,\psi}$: $\tilde{C} \times \tilde{D} \to \tilde{C} \times \tilde{D}$ such that $\mathbf{Wt}_{i,c,\psi}(\bar{c}; \bar{d})$ is undefined if $\prod_j(\bar{c}) \neq c$, otherwise $\mathbf{Wt}_{i,c,\psi}(\bar{c}; \bar{d}) = \bar{c}'; \langle \bar{d}'_1, \ldots, \bar{d}'_{n+1} \rangle$, where $\bar{c}' = \bar{c}$, $\bar{d}'_l = \bar{d}_l$ for $l \in [n]$ and

$$\begin{aligned} \bar{d}_{n+1} &= \bar{d}_{n+1} && \text{if } \psi_i(\bar{d}_j) \text{ does not hold.} \\ &= \bar{d}_{n+1}[z/i], \quad \text{where} \quad z = \prod_i (\mu^j(c; \bar{d}_j)) && \text{if } \psi_i(\bar{d}_j) \text{ holds.} \end{aligned}$$

We note that $\mathbf{Rd}_{i,c}$ and $\mathbf{Rd}_{i,c'}$ are identical instructions iff c and c' belong to the same process. The set $A_\psi(c)$ is $\{\boldsymbol{\lambda}_c, \mathbf{Rd}_{i,c}, \mathbf{Wt}_{i,c,\psi}\}_{i=1}^m$.

If $\bar{\sigma}$ is a finite sequence of primitive instructions, we can easily define a partial function on extended i.d.'s by induction on its length. If $\bar{\sigma} = (\sigma_1)$ then $\bar{\sigma}_1(\bar{I})$ is already given by Definition 7. If $\bar{\sigma} = \bar{\sigma}'; \bar{\sigma}_1$, then $\bar{\sigma}(\bar{I})$ is undefined if $\bar{\sigma}'(\bar{I})$ is undefined. Otherwise $\bar{\sigma}(\bar{I}) = \sigma_1(\bar{\sigma}'(\bar{I}))$.

DEFINITION 8. A sequence of primitive instructions of the form $\bar{\sigma} = (\mathbf{Rd}_{i_1,c}, \mathbf{Rd}_{i_2,c}, \ldots, \mathbf{Rd}_{i_m,c}, \mathbf{Wt}_{j_1,c,\psi}, \ldots, \mathbf{Wt}_{j_m,c,\psi}, \boldsymbol{\lambda}_c)$, where $\{i_1, \ldots, i_m\} = \{j_1, \ldots, j_m\} = [m]$, is called a *simple realization of* c *relative to* ψ provided for all extended i.d.'s $\bar{I} = \bar{c}; \langle \bar{d}_1, \ldots, \bar{d}_{n+1} \rangle$, where $\prod_j(\bar{c}) = c$, if $\bar{\sigma}(\bar{I}) = \bar{c}'; \langle \bar{d}'_1, \ldots, \bar{d}'_{n+1} \rangle$ then $\mu(c; \bar{d}_{n+1}) = \bar{d}'_{n+1}$ and $\lambda(c; \bar{d}_{n+1}) = \prod_j(\bar{c}')$.

We want to define some notion of an instruction c "depending" and "changing" a variable. Recalling the introductory example, one would expect c_0 to depend on t and to change x. Yet because there are different choices of the predicate set ψ, this has to be defined with care.

DEFINITION 9. Let $l \in [m]$, $\bar{I}^{(k)} = \bar{c}^{(k)}; \langle \bar{d}_1^{(k)}, \ldots, \bar{d}_{n+1}^{(k)} \rangle$ for $k = 1, 2, \ldots$ and let c belong to process j.

(i) $\boldsymbol{\lambda}_c$ *depends* on l iff there exist $\bar{d}_j^{(1)}$ and $\bar{d}_j^{(2)}$ which are identical except on l, $\boldsymbol{\lambda}_c(\bar{I}^{(1)}) = \bar{I}^{(3)}$, $\boldsymbol{\lambda}_c(\bar{I}^{(2)}) = \bar{I}^{(4)}$ and $\prod_j(\bar{c}^{(3)}) \neq \prod_j(\bar{c}^{(4)})$.

(ii) ψ_i *depends* on l iff there exist $\bar{d}$ and $\bar{d}'$ which are identical except on l, and $\psi_i(\bar{d})$ holds but $\psi_i(\bar{d}')$ does not.

(iii) $\mathbf{Wt}_{i,c,\psi}$ *depends* on l iff either ψ depends on l or there exist $\bar{d}$ and $\bar{d}'$ which are identical except on l, $\psi_i(\bar{d})$ holds and $\prod_i(\mu^j(c; \bar{d})) \neq \prod_i(\mu^j(c; \bar{d}'))$.

(iv) $\mathbf{Wt}_{i,c,\psi}$ *changes* l iff $l = i$ and for some $\bar{d}$, $\psi_i(\bar{d})$ holds.

Let $\bar{\sigma} = (\mathbf{Rd}_{i_1,c}, \ldots, \mathbf{Rd}_{i_m,c}, \mathbf{Wt}_{j_1,c,\psi}, \ldots, \mathbf{Wt}_{j_m,c,\psi}, \lambda_c)$ be a simple realization of c relative to ψ.

(v) $\bar{\sigma}$ *depends* on l iff λ_c depends on l or some $\mathbf{Wt}_{j,c,\psi}$ depends on $l, j \in [n]$.

(vi) $\bar{\sigma}$ *changes* l if some $\mathbf{Wt}_{j,c,\psi}$ changes l.

There are some subtleties in Definition 9 (which, hopefully, are inherent rather than an artifact of our formulation). Consider part (iii) of the definition applied to both the actions.

$$\mathbf{Wt}_{x,c_0}\colon x \leftarrow f(x', t')$$

and

$$\mathbf{Wt}_{x,c_0,t=0}\colon \quad \textit{if} \quad t' = 0 \quad \textit{then} \quad x \leftarrow 0 \; \textit{fi}$$

of the introductory example. We see that $\mathbf{Wt}_{x,c_0}$ is dependent on x because the implicit predicate ψ_x here is "*true*" and if we pick $\bar{d} = (x, t)$ and $\bar{d}' = (x', t')$ such that $\bar{d}$ and $\bar{d}'$ are identical except on x, i.e., $t = t'$, $x \neq x'$, then $\prod_x(\mu(c_0, \bar{d})) \neq \prod_x(\mu(c_0, \bar{d}'))$. However, $\mathbf{Wt}_{x,c_0,t=0}$ is *not* dependent on x because the predicate ψ_x is "$t = 0$" and if $\bar{d} = (x, t)$ and $\bar{d}' = (x', t')$ are identical except on x, either both $\psi_x(\bar{d})$ and $\psi_x(\bar{d}')$ hold or both do not hold. If they hold, then $\prod_x(\mu(c_0, \bar{d})) = \prod_x(\mu(c_0, \bar{d}')) = (0, 0)$ since $t = 0$ and x is set to 0. Hence depending on the choice of ψ, the instruction c_0 may be regarded as dependent or independent of x.

Additional Remark. We may say that instruction c *inherently depends* on variable i iff for all ψ, if $\bar{\sigma}$ is any simple realization of c relative to ψ, then $\bar{\sigma}$ depends on i. The above discussion shows that c_0 does not inherently depend on x. But it is not hard to convince oneself that c_0 inherently depends on t.

Let Σ be the system of processes as before. Let Γ be a function on $\bigcup_{j=1}^{n} C^j$ which assigns a simple realization $\Gamma(c)$ to each $c \in \bigcup_{j=1}^{n} C^j$. There is no restriction that the predicate set ψ for different instructions c has to be related. We call the pair $\langle \Sigma, \Gamma \rangle$ an *implemented system*, which is fixed for the rest of this paper. We easily extend Γ to sequences of instructions: If $\bar{c} = (c_1, \ldots, c_k)$ then $\Gamma(\bar{c}) = \Gamma(c_1); \Gamma(c_2); \ldots; \Gamma(c_k)$. If $\bar{c}$ is the null sequence $\Gamma(\bar{c})$ is also null.

Let ξ be a function from $[n]$ into the nonnegative numbers, and let C^* be a set of sequences of instructions $\{\bar{c}^1, \bar{c}^2, \ldots, \bar{c}^n\}$, where each $\bar{c}^j = (c_1{}^j, \ldots, c^j_{\xi(j)})$. If $\xi(j) = 0$, then $\bar{c}^j$ is null. C^* and ξ will be fixed from now on.

DEFINITION 10. A *simultaneous realization of* C^* is an element of the shuffle product of $\{\Gamma(\bar{c}^1), \ldots, \Gamma(\bar{c}^n)\}$. A sequence of primitive instructions is said to be a simultaneous realization if there exists a set C^* for which it is the simultaneous realization.

DEFINITION 11. We define the binary relation on i.d.'s, $\rightarrow_{(s)}$, such that $I \rightarrow_{(s)} I'$ holds iff there exists $C^* = \{\bar{c}^i\}_{i=1}^{n}$ a set of sequences of instructions, $\bar{\sigma}$ a simultaneous realization of C^* and $\bar{I}, \bar{I}'$ two extended i.d.'s such that

(i) $\bar{\sigma}(\tilde{I})$ is defined and $\bar{\sigma}(\tilde{I}) = \tilde{I}'$.

(ii) $E(\tilde{I}) = I$ and $E(\tilde{I}') = I'$.

$I \to_{(s)} I'$ is to be read "I simultaneously derives I'." Extending our previous terminology we say that C^* *acted simultaneously* in the transition $I \to_{(s)} I'$, and write $I \to_{(s)}^{C^*} I'$ or $I \to_s^{(\bar{\sigma})} I'$.

DEFINITION 12. A sequence of instructions $\bar{c} = (c_1, c_2, \ldots, c_l)$, where $l = \sum_{i=1}^{n} \xi(i)$, is a *commutative realization of* C^* iff $\bar{c}$ is an element of the shuffle product of the sequences in C^*. If I and I' are i.d.'s such that $\bar{c}(I) = I'$, then we write $I \to_{(c)}^{C^*} I'$ (or, $I \to_{(c)} I'$) to be read "I commutatively derives I'," and C^* *acted commutatively* in the transition $I \to_{(c)} I'$.

It seems clear that $I \to_{(c)}^{C^*} I'$ implies $I \to_{(s)}^{C^*} I'$. But the converse is false in general. Our main result concerns discovering conditions under which the converse holds. That is, for all C^*, $I \to_{(s)}^{C^*} I'$ iff $I \to_{(c)}^{C^*} I'$. When this obtains, we describe the situation with the convenient slogan:

"simultaneity $\equiv$ commutativity."

The significance of this condition is that modeling simultaneity by commutativity is then justified.

5. THE MAIN RESULT

It seems clear that the attainment of "simultaneity $\equiv$ commutativity" comes from restrictions on the power of instructions. For this we need some further definitions.

DEFINITION 13. For each c, the *range* of c is $\rho(c) = \{i \in [m]: \Gamma(c) \text{ changes } i\}$ and the *domain* of c is $\delta(c) = \{i \in [m]; \Gamma(c) \text{ depends on } i\}$.

DEFINITION 14. The variable i, $i \in [m]$, is said to be *private to process* j, $j \in [n]$, iff for all instructions c, $i \in \rho(c) \cup \delta(c)$ implies $c \in C^j$. If a variable is not private to any process, then it is *public*.

DEFINITION 15. An instruction $c \in C^i$ is a *load-type* instruction iff $\delta(c)$ contains at least one public variable. It is a *store-type* instruction if $\rho(c)$ contains at least one public variable.

Private variables are those that are accessed (read or written) by at most one process. If the class of load-type instructions and the class of store-type instructions are disjoint, we say that instructions are *dichotomized*. Note that when instructions are dichotomized, then each instruction falls under exactly one of the descriptions of a load-type or a store-type or uses only private variables.

We can now state our main result.

THEOREM (simultaneity ≡ commutativity). *Under the conditions that*

(i) *instructions are dichotomized and*

(ii) $\forall$ *instructions* c, *at most one variable in* $\delta(c) \cup \rho(c)$ *is public,*

then "simultaneity ≡ commutativity."

The following two lemmas assume the conditions of the theorem.

LEMMA 1. *Let* $\tilde{I}^{(i)} = \bar{c}^{(i)}; \langle \bar{d}_1^{(i)}, \ldots, \bar{d}_{n+1}^{(i)} \rangle$ *for* $i = 1, 2, \ldots$. *If* $E(\tilde{I}^{(1)}) = E(\tilde{I}^{(2)})$, *then for all simultaneous realizations* $\bar{\sigma}$, $\bar{\sigma}(\tilde{I}^{(1)})$ *is defined iff* $\bar{\sigma}(\tilde{I}^{(2)})$ *is defined. Moreover,* $\bar{\sigma}(\tilde{I}^{(1)}) = \tilde{I}^{(3)}$ *and* $\bar{\sigma}(\tilde{I}^{(2)}) = \tilde{I}^{(4)}$ *implies that* $E(\tilde{I}^{(3)}) = E(\tilde{I}^{(4)})$.

Proof. It is sufficient to show that for any $\bar{\sigma}'$, prefix of $\bar{\sigma}$, such that $\bar{\sigma}'(\tilde{I}^{(1)}) = \tilde{I}^{(5)}$ and $\bar{\sigma}'(\tilde{I}^{(2)}) = \tilde{I}^{(6)}$ (if defined), we have

(*) $E(\tilde{I}^{(5)}) = E(\tilde{I}^{(6)})$, whenever one is defined.

Let $\bar{\sigma}'$ be $\bar{\sigma}''; e$ for $\bar{\sigma}''$ a prefix of $\bar{\sigma}$ and e a single primitive instruction. Let $\bar{\sigma}''(\tilde{I}^{(1)}) = \tilde{I}^{(7)}$, $\bar{\sigma}''(\tilde{I}^{(2)}) = \tilde{I}^{(8)}$ (if defined). By induction hypothesis $E(\tilde{I}^{(7)}) = E(\tilde{I}^{(8)})$ if either one is defined. If both $\tilde{I}^{(7)}$ and $\tilde{I}^{(8)}$ are undefined, then both $\tilde{I}^{(5)}$ and $\tilde{I}^{(6)}$ are undefined and we are done. So assume $\tilde{I}^{(7)}$ and $\tilde{I}^{(8)}$ are defined. It is straightforward to check for each of the three cases of $e = \boldsymbol{\lambda}_c$ or $e = \mathbf{Rd}_{i,c}$ or $e = \mathbf{Wt}_{i,c,\psi}$ that $E(\tilde{I}^{(7)}) = E(\tilde{I}^{(8)})$ implies (*) for $\bar{\sigma}'$. Q.E.D.

Lemma 1 says that if $\tilde{I}^{(1)}$ and $\tilde{I}^{(2)}$ are two extended i.d.'s such that their counter values and external data are identical, then any simultaneous realization that transforms them (if defined) does it in a way that the counter values and external data of the transformed extended i.d.'s again agree.

We say that $\mathbf{Wt}_{i,c,\psi}$ *accesses public data* if i is public. We say that $\mathbf{Rd}_{i,c}$ *accesses public data* if i is public and for some j, $\mathbf{Wt}_{j,c,\psi}$ depends on i, No other instructions access public data.

LEMMA 2. *Let* σ_1 *and* σ_2 *be primitive instructions belonging to different processes such that at most one of them accesses public data. Let* $\bar{\tau} = \bar{\tau}^{(1)}; \sigma_1; \sigma_2; \bar{\tau}^{(2)}$ *and* $\bar{\tau}' = \bar{\tau}^{(1)}; \sigma_2; \sigma_1; \bar{\tau}^{(2)}$ *be simultaneous realizations. Then for all* $\tilde{I}$, $\bar{\tau}(\tilde{I})$ *is defined iff* $\bar{\tau}'(\tilde{I})$ *is defined. Moreover* $E(\bar{\tau}(\tilde{I})) = E(\bar{\tau}'(\tilde{I}))$ *whenever defined.*

Proof. By symmetry, let us assume $\bar{\tau}(\tilde{I})$ is defined. Then $(\bar{\tau}^{(1)}; \sigma_1; \sigma_2)(\tilde{I})$ is defined. To be specific, let σ_1 be in process 1, σ_2 in process 2. Since σ_1 cannot affect the counter value of process 2, and σ_2 cannot affect the counter value of process 1, it is easy to see that $(\bar{\tau}^{(1)}; \sigma_2; \sigma_1)(\tilde{I})$ is defined. We now claim for all prefixes $\bar{\tau}^{(3)}$ of $\bar{\tau}^{(2)}$, if $\bar{\tau}^{(4)} = \bar{\tau}^{(1)}; \sigma_1; \sigma_2; \bar{\tau}^{(3)}$ and $\bar{\tau}^{(5)} = \bar{\tau}^{(1)}; \sigma_2; \sigma_1; \bar{\tau}^{(3)}$ then

(i) $\bar{\tau}^{(4)}(\tilde{I})$ and $\bar{\tau}^{(5)}(\tilde{I})$ are defined,

(ii) $E(\bar{\tau}^{(4)}(\tilde{I})) = E(\bar{\tau}^{(5)}(\tilde{I}))$.

This claim would establish our lemma. We use induction on the length of $\bar{\tau}^{(3)}$. If $\bar{\tau}^{(3)}$ is the null sequence, our remarks above show (i). To see (ii), it is easy to see that the counter values in $\bar{\tau}^{(4)}(\tilde{I})$ and $\bar{\tau}^{(5)}(\tilde{I})$ are identical when $\bar{\tau}^{(3)}$ is empty. If both σ_1 and σ_2 are not writing instructions, then the external data in $\bar{\tau}^{(4)}(\tilde{I})$ and $\bar{\tau}^{(5)}(\tilde{I})$ are identical, being both equal to the external data of $\tilde{I}$. So assume σ_2 is $\mathbf{Wt}_{i,c,\psi}$ for some i. Surely the internal data of process 2 in $(\bar{\tau}^{(1)}; \sigma_1)(\tilde{I})$ and $\bar{\tau}^{(1)}(\tilde{I})$ are identical. Hence the ith variable of the external data of $(\bar{\tau}^{(1)}; \sigma_1; \sigma_2)(\tilde{I})$ and $(\bar{\tau}^{(1)}; \sigma_2)(\tilde{I})$ are identical. If σ_1 is not a writing instruction, then this would imply the external data of $(\bar{\tau}^{(1)}; \sigma_2; \sigma_1)(\tilde{I})$ equals those of $(\bar{\tau}^{(1)}; \sigma_2)(\tilde{I})$ and $(\bar{\tau}^{(1)}; \sigma_1; \sigma_2)(\tilde{I})$ and we are done. So let σ_1 be a writing instruction of the form $\mathbf{Wt}_{j,c',\psi'}$. We may surely assume both ψ and ψ' are not "**false**," otherwise the writing action with the false predicate does nothing and we are back to the previous cases. But then $j \neq 1$, otherwise $i = j$ would be public contradicting our assumptions on σ_1, σ_2. Again we see the external data of $(\bar{\tau}^{(1)}; \sigma_1; \sigma_2)(\tilde{I})$ and $(\bar{\tau}^{(1)}; \sigma_2; \sigma_1)(\tilde{I})$ agree. This concludes the basis case. The rest of the inductive step is similar, but is tedious and unenlightening, so we omit it. Q.E.D.

Lemma 2 may easily be strengthened in various ways if desired. It permits us to "commute" adjacent primitive instructions without affecting the final outcome. This lemma is used repeatedly in the main proof.

Proof of Theorem. One direction of the theorem is easily derived without even using the restrictions (i) and (ii) on instructions. Let $\tilde{I}^{(i)} = \bar{c}^{(i)}; \langle \vec{d}_1^{(i)}, \ldots, \vec{d}_{n+1}^{(i)} \rangle$ and $I^{(i)} = E(\tilde{I}^{(i)})$ for $i = 1, 2, \ldots$. If $\bar{c}$ is a commutative realization of C^* and $\bar{c}(I^{(1)}) = I^{(2)}$, we want to show that there is a simultaneous realization of C^*, $\bar{\sigma}$, such that $I^{(1)} \to^{\bar{\sigma}}_{(s)} I^{(2)}$. Let $\bar{c} = (c_1, \ldots, c_l)$, $l = \sum_{i=1}^{n} \xi(i)$. If $l = 1$, then the result follows by definition of a simple realization. If $l > 1$, assume the result inductively for $l - 1$. Note that $\bar{c}' = (c_1, \ldots, c_{l-1})$ is a commutative realization of some other set of sequences of instructions. Let $I^{(1)} \to^{\bar{c}}_{(c)} I^{(3)}$ and $I^{(3)} \to^{c_l} I^{(2)}$ for some $I^{(3)}$. By induction $\exists \bar{\sigma}'$, a simultaneous realization such that $I^{(1)} \to^{\bar{\sigma}'} I^{(3)}$. This means that $\bar{\sigma}'(\tilde{I}^{(1)}) = \tilde{I}^{(3)}$. From Lemma 1 $I^{(3)} \to^{c_l} I^{(2)}$ implies that there exists a simple realization of c_l, $\bar{\sigma}''$, such that $\bar{\sigma}''(\tilde{I}^{(3)}) = \tilde{I}^{(2)}$. Hence $(\bar{\sigma}'; \bar{\sigma}'')(\tilde{I}^{(1)}) = \tilde{I}^{(2)}$. Hence $\bar{\sigma} = \bar{\sigma}'; \bar{\sigma}''$ will be the required simultaneous realization of C^* such that $I^{(1)} \to^{\bar{\sigma}}_{(s)} I^{(2)}$.

The other direction of the theorem makes essential use of the conditions (i) and (ii) in the theorem. The proof proceeds by induction on $l = \sum_{i=1}^{n} \xi(i)$. If $l = 1$, the result is already given by the definition of a simple realization. So let $l > 1$ and assume the result for $l - 1$. Let $\bar{\sigma} = (\sigma_1, \ldots, \sigma_k)$ be a simultaneous realization of C^*, and h be the largest index such that σ_h accesses public data. Let σ_h belong to the instruction c. We first claim that $\bar{\sigma}$ may be transformed (by permutation) into $\bar{\sigma}^{(1)} = (\sigma_1^{(1)}, \ldots, \sigma_k^{(1)})$ such that for all $\tilde{I}$, $\bar{\sigma}^{(1)}(\tilde{I})$ is defined iff $\bar{\sigma}(\tilde{I})$ is defined, $\bar{\sigma}^{(1)}(\tilde{I}) = \bar{\sigma}(\tilde{I})$ whenever defined, and if h_1 is the largest index such that $\sigma_{h_1}^{(1)}$ accesses public data, then $\sigma_{h_1}^{(1)} = \sigma_h$ and for all $i > h_1$, $\sigma_i^{(1)}$ belongs to c. This is done by applying Lemma 2 repeatedly: If for all $i > h$, σ_i belongs to c, then we are done. Otherwise, pick the smallest index, i, such that $i > h$, and σ_i does not belong to c. By choice of h, σ_i does not access public data, so we may move σ_i to the position immediately to the left of σ_h. It is easy to see that this procedure must always halt and the final sequence has the desired property. We now claim that $\bar{\sigma}^{(1)}$ may be

transformed into $\bar{\sigma}^{(2)} = (\sigma_1^{(2)},\ldots, \sigma_k^{(2)})$ such that for some $g \in [k]$, $\bar{\sigma}^{(2)} = \bar{\sigma}^{(3)}; \bar{\sigma}^{(4)}$, $\bar{\sigma}^{(3)} = (\sigma_1^{(2)},\ldots, \sigma_g^{(2)})$ and $\bar{\sigma}^{(4)} = (\sigma_{g+1}^{(2)},\ldots, \sigma_k^{(2)})$, where $\bar{\sigma}^{(2)}(\tilde{I}) = \bar{\sigma}^{(1)}(\tilde{I})$ whenever one of $\bar{\sigma}^{(2)}(\tilde{I})$ or $\bar{\sigma}^{(1)}(\tilde{I})$ is defined, $\bar{\sigma}^{(4)}$ is $\Gamma(c)$ and $\bar{\sigma}^{(3)}$ is the simultaneous realization of some set of instruction sequences. Again this is done by repeated application of Lemma 2. We do this by moving all the primitive instructions that belong to c to the right of primitive instructions that do not belong to c. Note that the primitive instructions that belong to c which have to be moved to the right *cannot* be $\sigma_{h_1}^{(1)}$ by construction of $\bar{\sigma}^{(1)}$. By the assumption that instructions are dichotomized, such primitive instructions cannot access the same public variable as $\sigma_{h_1}^{(1)}$. Since one would have to be a reading, the other a writing primitive instruction. Since there is only 1 public variable in c, such primitive instructions accesses no public variables. Lemma 2 is then applicable.

Let $\tilde{I}^{(i)} = \bar{c}^{(i)}; \langle \vec{d}_1^{(i)},\ldots, \vec{d}_{n+1}^{(i)} \rangle$ and $I^{(i)} = E(\tilde{I}^{(i)})$ for $i = 1, 2,\ldots$. Given that $I^{(1)} \to_{(s)}^{\bar{\sigma}} I^{(2)}$, we want to show that $I^{(1)} \to_{(c)}^{\bar{c}'} I^{(2)}$ for some commutative realization of C^*, $\bar{c}$. From our transformation of $\bar{\sigma}$ into $\bar{\sigma}^{(2)}$, we get that $I^{(1)} \to_{(s)}^{\bar{\sigma}^{(2)}} I^{(2)}$. This means that $I^{(1)} \to_{(s)}^{\bar{\sigma}^{(3)}} I^{(3)} \to_{(s)}^{\bar{\sigma}^{(4)}} I^{(2)}$ for some $I^{(3)}$, since $\bar{\sigma}^{(2)} = \bar{\sigma}^{(3)}; \bar{\sigma}^{(4)}$. By the induction hypothesis, $\exists$ a commutative realization $\bar{c}'$, such that $I^{(1)} \to_{(c)}^{\bar{c}'} I^{(3)}$. Therefore, choosing $\bar{c} = \bar{c}'; c$ we obtain $I^{(1)} \to_{(c)}^{\bar{c}} I^{(2)}$, as required. Q.E.D.

6. Counterexamples

The examples of this section show that our result is tight in the sense that neither of its assumptions can be omitted. First, we show that the particular implementation of instructions (as specified by Γ) is crucial to our result. Consider the example shown in Fig. 4. We may implement c_0^1 as $\Gamma(c_0^1) = (\mathbf{Rd}_{X,c_0^1}, \mathbf{Wt}_{X,c_0^1,\mathbf{false}})$ in which case c_0^1 does not depend on the variable X. The implemented system then satisfies the conditions of the main theorem. The final value of X is always 1. On the other hand, if c_0^1 is implemented as $\Gamma(c_0^1) = (\mathbf{Rd}_{X,c_0^1}, \mathbf{Wt}_{X,c_0^1,\mathbf{true}})$ then c_0^1 depends on X. This violates the dichotomy of instructions since c_0^1 also changes X. The final value of X in this case could be 0.

INITIALLY $X = 0$

(Process 1)	c_0^1 :	$X \leftarrow X$
	c_1^1 :	*halt*
(Process 2)	c_0^2 :	$X \leftarrow 1$
	c_1^2 :	*halt*

FIGURE 4

INITIALLY $(X, Y) = (0, 0)$

(Process 1)	c_0^1 :	$(X, Y) \leftarrow (1, 0)$
	c_1^1 :	*halt*
(Process 2)	c_0^2 :	$(X, Y) \leftarrow (0, 1)$
	c_1^2 :	*halt*

FIGURE 5

INITIALLY $(X, Y) = (0, 0)$

(Process 1)	c_0^1 :	$U \leftarrow X + Y$
	c_1^1 :	*halt*
(Process 2)	c_0^2 :	$X \leftarrow 1$
	c_1^2 :	$Y \leftarrow 2$
	c_2^2 :	*halt*

FIGURE 6

Returning to the example of Fig. 2, we see that the instruction c_0^1: $S \leftarrow S + 1$ violates the dichotomy of instructions since c_0^1 is both a load-type and a store-type instruction. As predicted by the theorem, commutativity $\not\equiv$ simultaneity for this system. This example is almost identical to that of Fig. 4, the only difference being that c_0^1 in Fig. 4 does not inherently depend on X, but c_0^1 in Fig. 2 inherently depends on S (that is why we need not show explicitly any simple realizations of $S \leftarrow S + 1$).

Consider the system shown in Fig. 5. We assume that ψ_X and ψ_Y are **true**. Commutativity alone can get a final value of (1, 0) or (0, 1) for (X, Y). But simultaneity may result in (0, 0) and/or (1, 1) also depending on which simple realizations of c_0^1 and c_0^2 we choose. The different results come from choices of whether to write X or Y first in c_0^1 and c_0^2. Note that in this case, the instructions are dichotomized but the single public variable constraint is violated by writing into two public variables simultaneously. The reader may feel that assigning to more than one public variable simultaneously may be unusual for many programming languages. The next example (Fig. 6) is more natural in that sense. It violates the single public variable constraint in a different manner: by reading from two public variables simultaneously instead of writing into two public variables simultaneously. In this example, U is a private variable. Assuming commutativity U may finally be 0, 1, or 3. If we choose the simple realization $\Gamma(c_0^1) = (\mathbf{Rd}_{X,c_0^1}, \mathbf{Rd}_{Y,c_0^1}, \mathbf{Rd}_{U,c_0^1}, \mathbf{Wt}_{X,c_0^1,\psi}, \mathbf{Wt}_{Y,c_0^1,\psi}, \mathbf{Wt}_{U,c_0^1,\psi})$, where ψ_X and ψ_Y are **false** and ψ_U is **true**, then it is possible for U to attain a value of 2 in the simultaneity model.

7. CONCLUSIONS

We have introduced a formal model of a synchronization system, and within the model, formulated a precise notion of simultaneity. The main results show sufficient and nondeletable conditions for the formula "commutativity $\equiv$ simultaneity" to obtain. We have ignored failure functions in the main results, but it is not difficult to incorporate failure functions into the main result, treating the failure as a special type of instruction.

ACKNOWLEDGMENTS

We are happy to acknowledge the help of C. C. Elgot, with whom we had many substantive discussions during our early work on this paper. We thank Leslie Lamport for comments which led to the strengthening of our theorem from that in [11, 12], and we thank G. L. Peterson for his comments which led to a correction of our concept of "depends."

REFERENCES

1. A. J. BERNSTEIN, Analysis of programs for parallel processing, *IEEE Trans. Electronic Computers*, EC-15 (1966), 757–763.
2. A. CREMERS AND T. N. HIBBARD, "An Algebraic Approach to Concurrent Programming Control and Related Complexity Problems," Report, USC Computer Science Program, November, 1975.
3. E. W. DIJKSTRA, Solution of a problem in concurrent programming control, *Comm. ACM* 8 (1965), 569.
4. E. W. DIJKSTRA, Co-operating sequential processes, *in* "Programming Languages" (F. Genuys, Ed.), pp. 43–112, New York, Academic Press (1968).
5. P. GILBERT AND W. J. CHANDLER, Interference between communicating parallel processes, *Comm. ACM* 15, No. 6 (1972), 427–437.
6. R. M. KARP AND R. E. MILLER, Parallel program schemata, *J. Comput. System Sci.* 3 (1969), 147–195.
7. L. LAMPORT, "On Concurrent Reading and Writing," Report CA-7409-0511, Massachusetts Computer Assoc., Inc., September 1974; revised March 1976.
8. L. LAMPORT, "Time, Clocks and the Ordering of Events in a Distributed System," Report CA-7603-2911, Massachusetts Computer Assoc., Inc., March 1976.
9. R. J. LIPTON, "On Synchronization Primitive Systems," Ph. D. Thesis, Carnegie–Mellon University, 1973, and Research Report No. 22, Yale University, Department of Computer Science, October 1973.
10. R. E. MILLER, Relationships among models of parallelism and synchronization, *in* "Proceedings, Symposium on Petri Nets and Related Methods, July 1975," to appear.
11. R. E. MILLER AND C. K. YAP, "Formal Specification and Analysis of Loosely Connected Processes," IBM Research Report RC-6716, September, 1977.
12. R. E. MILLER AND C. K. YAP, On formulating simultaneity for studying parallelism and synchronization, *in* "Proceedings, Tenth Annual ACM Symposium on Theory of Computing, May 1, 1978," pp. 105–113.
13. G. L. PETERSON AND M. J. FISCHER, Economical solutions for the critical section problem in a distributed system, extended abstract, *in* "Proceedings, Ninth Annual ACM Symposium on Theory of Computing, May 1977," pp. 91–97.
14. R. L. RIVEST AND V. R. PRATT, The mutual exclusion problem for unreliable processes: Preliminary report, *in* "Proceedings, 17th Annual IEEE Symposium on Foundations of Computer Science, October 1976," pp. 1–8.
15. C. K. YAP, On abstract synchronization problems and synchronization systems, unpublished manuscript, 1976.
16. P. ZAVE, On the formal definition of processes, *in* "Proceedings, International Conference on Parallel Processing, 1976."
17. P. ZAVE AND D. R. FITZWATER, "Specification of Asynchronous Interactions Using Primitive Functions," Technical Report, Dept. of Computer Science, University of Maryland, 1977.